The Emergence of Software Diversity in Maven Central

César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, Benoit Baudry

The Emergence of Software Diversity in Maven Central

César Soto-Valero†, Amine Benelallam*, Nicolas Harrand†, Olivier Barais*, and Benoit Baudry†
†*KTH Royal Institute of Technology, Stockholm, Sweden*
Email: {cesarsv, harrand, baudry}@kth.se
**Univ Rennes, Inria, CNRS, IRISA, Rennes, France*
Email: amine.benelallam@inria.fr, barais@irisa.fr

***Abstract*—Maven artifacts are immutable: an artifact that is uploaded on Maven Central cannot be removed nor modified. The only way for developers to upgrade their library is to release a new version. Consequently, Maven Central accumulates all the versions of all the libraries that are published there, and applications that declare a dependency towards a library can pick any version. In this work, we hypothesize that the immutability of Maven artifacts and the ability to choose any version naturally support the emergence of software diversity within Maven Central. We analyze 1,487,956 artifacts that represent all the versions of 73,653 libraries. We observe that more than 30% of libraries have multiple versions that are actively used by latest artifacts. In the case of popular libraries, more than 50% of their versions are used. We also observe that more than 17% of libraries have several versions that are significantly more used than the other versions. Our results indicate that the immutability of artifacts in Maven Central does support a sustained level of diversity among versions of libraries in the repository.**

***Index Terms*—Maven Central, Software Diversity, Library Versions, Evolution, Open-Source Software**

I. Introduction

Maven Central is the most popular repository to distribute and reuse JVM-based artifacts (i.e., reusable software packages implemented in Java, Clojure, Scala or other languages that can compile to Java bytecode). By September 6, 2018, Maven Central contains over $2.8M$ artifacts and serves over $100M$ downloads every week [1]. The Maven dependency management system, which is able to resolve transitive dependencies automatically, has been key to this success: it relieves developers from the complexity of manual management of their dependencies. Uploading artifacts into Maven Central is the most effective way for open source projects to remain permanently accessible to their users. In this way, every build tool able to download Java libraries can fetch from a world of libraries and dependencies in a single and authoritative place.

In this work, we analyze software artifacts from the perspective of one essential characteristic enforced by Maven Central: immutability[1]. All artifacts (code packages, documentation, dependency declarations, etc.) that are uploaded on Maven Central are immutable: they cannot be rewritten nor deleted. This is a critical design choice that has a significant influence on the way the Maven Central repository is utilized. We hypothesize that this design decision is a great opportunity to prevent dependency monoculture [2] and increase the diversity [3] among software dependencies.

Previous works have analyzed Maven artifacts from the perspective of the risks induced by immutability. First, the redundancy in multiple versions can introduce conflicts among dependencies, e.g., trying to load the same class several times. This risk has been extensively analyzed by Wang and colleagues [4]. Second, the projects that depend on a library need to explicitly update their dependency descriptions in order to benefit from the update. This represents a risk since these projects can eventually rely on outdated dependencies [5] that can contain security issues [6] or API breaking changes [7].

We take a fresh look at the presence of multiple versions of the same library in Maven Central, and consider it as an opportunity. We analyze how the ability to choose any library version for software reuse supports the emergence of software diversity in the repository and how this diversity of versions fuels the success of popular libraries. We consider this emergent diversity of reused versions as an opportunity since it participates in mitigating the risks of software monoculture [8]. Overcoming this type of monoculture is essential to build resilient and robust software systems [3], [9], [10].

To conduct this empirical study, we rely on an existing dataset, the Maven Dependency Graph [1], which captures a snapshot of Maven Central as of September 6, 2018. This dataset comes in the form of a temporal graph with metadata of $2.4M$ artifacts belonging to $223K$ libraries, with more than $9M$ direct dependency relationships between them. In order to enable reasoning not only at the artifact level but also at the library level, we extend this dataset with another abstraction layer capturing dependencies at the library level.

We measure activity, popularity and timeliness of a subset of $73,653$ libraries with multiple versions, which represents 61.81% of the total number of artifacts in Maven Central. We empirically investigate whether the diversity of library versions is a valuable design choice. Our contributions are as follows:

- a quantitative analysis of the diversity of usage and popularity of library versions;
- evidence of the presence of large quantities of artifacts that participate in the emergence of diversity;
- open science with replication code and scripts available online.

[1]Sonatype community support: https://issues.sonatype.org/browse/OSSRH-39131

II. Background and Definitions

In this section, we describe the dataset of Maven artifacts that constitutes the raw material for our work, as well as its extended library-level abstraction.

A. The Maven Dependency Graph

To conduct this empirical study, we rely on the Maven Dependency Graph (MDG), a dataset that captures all of the artifacts deployed on the Maven Central repository as of September 6, 2018 [1]. The MDG includes $2,407,335$ artifacts. Each artifact is uniquely identified with a triplet (*'groupId:artifactId:version'*). The *groupId* identifier is a way of grouping different Maven artifacts, for instance by library vendor. The *artifactId* identifier refers to the library name. Finally, the *version* identifies each library release uniquely. For example, the triplet *'org.neo4j:neo4j-io:3.4.7'* identifies the version 3.4.7 of an input/output abstraction layer for the Neo4j graph database. The MDG also includes $9,715,669$ dependency relationships as declared in the Project Object Model (`pom.xml`) file of each artifact.

In this work, we focus on *libraries*, i.e., the sets of artifacts that share the same tuple '*groupId:artifactId*' but have different versions. The MDG includes $223,478$ of such libraries, but the concept of library is not rigorously captured in the graph. Consequently, we extend the artifact nodes of the MDG with labels referring to their corresponding library. We call `LIBS` the set of all libraries in Maven Central. We introduce an ordering function denoted $<_v$ that leverages the standard version numbering policy described by the Apache Software Foundation[2] in order to compare the different versions of artifacts belonging to the same library. For instance, 1.2.0 $<_v$ 2.0.0. We also define a temporal ordering function denoted by $<_t$ to compare the release dates of different artifacts. For example, '12-09-2011' $<_t$ '30-03-2015'. In the remainder of the paper, we refer to artifacts as *library versions* or simply *versions*. We define the MDG as follows:

Definition 1. ***Maven Dependency Graph.*** *The MDG is a vertex-labelled graph, where vertices represent Maven library versions, and edges describe dependency relationships or precedence relationships. We use a labeling function over vertices to group versions by library. We define the MDG as* $\mathcal{G} = (\mathcal{V}, \mathcal{D}, \mathcal{N}, \mathcal{L}, \mathcal{R})$*, where,*

- *the set of vertices* $\mathcal{V}$ *represents the library versions present in Maven Central*
- *the set of directed edges* $\mathcal{D}$ *represents dependency relationships between library versions*
- *the set of directed edges* $\mathcal{N}$ *represents versions precedence relationships, where the version of the source node is strictly lower than the version of the target node w.r.t.* $<_v$
- *the surjective labelling function* $\mathcal{L}$ *returns the corresponding library of a given library version* $v \in \mathcal{V}$*, defined as* $\mathcal{L} : \mathcal{V} \rightarrow \texttt{LIBS}$

[2]https://cwiki.apache.org/confluence/display/MAVEN/Version+number+policy

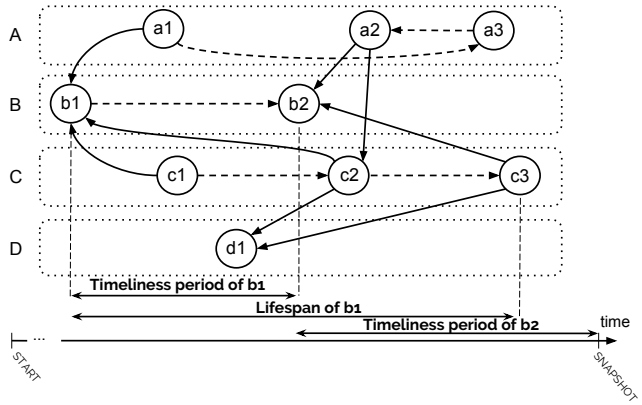

Fig. 1. Example of relationships between library versions in the Maven Dependency Graph.

- *the temporal function* $\mathcal{R}$ *refers to the date at which a library version* $v \in \mathcal{V}$ *was deployed, defined as* $\mathcal{R} : \mathcal{V} \rightarrow T$*, where* T *is a* $<_t$*-ordered discrete time domain*

In the MDG, T is bounded to ['15-05-2002', '06-09-2018'], where the lower bound refers to the date when the first library was deployed on Maven Central. In the rest of the paper, we refer to the lower and upper bounds respectively as `START` and `SNAPSHOT`, and we use days as the time granularity.

Figure 1 illustrates the different nodes and relationships within a simplified graph $\mathcal{G}$ composed of four libraries (A,B,C,D) and nine library versions $\{a_1, a_2, a_3, b_1, b_2, c_1, c_2, c_3, d_1\}$. The regular edges represent dependency relationships. For example, the first version of A (a_1) depends on the first version of B (b_1), and the second version of A (a_2) depends on the second version of B (b_2) and C (c_2). The dashed edges represent precedence relationships, and all vertices that are related through such edges constitute the different versions of a library. In Figure 1, we place nodes in a temporal order, from left to right, corresponding to the deployment date, thus the node b_1 is the firstly deployed, while the node c_3 is the most recently deployed.

The temporal order of releases does not imply a similar versioning order for a given library. In some cases, library instances with lower version number may be released after library versions with a greater version number, e.g., in case of a library version downgrade or maintenance of several major library versions. In Figure 1, we can see that $a_2 <_t a_3$ and $a_3 <_v a_2$. Note, this is a common practice adopted by very popular libraries such as Apache CXF[3], and Mule[4] [11].

Definition 2. ***Additional notations.*** *For further references in the MDG, we introduce the following notations:*

- $next(v)$*: the next release of a given library version* v *w.r.t. the ordering function* $<_v$
- $next_{all}(v)$*: transitive closure on the next releases of a library version* v

[3]https://cxf.apache.org

[4]https://www.mulesoft.com

- *latest: the library version v such that $\nexists\ next(v)$*
- *LATESTS: the set of all latest library versions in a dependency graph $\mathcal{G}$*
- *$deps(v)$: $\mathcal{V} \rightarrow \mathcal{V}^n$, with $n \in \mathbb{N}$: the set of direct dependencies of a given library version $v \in \mathcal{V}$*
- *$deps_{tree}(v)$: the whole dependency tree of v*
- *$users(v)$: $\mathcal{V} \rightarrow \mathcal{V}^n$, with $n \in \mathbb{N}$: the set of library versions declaring a dependency towards v*
- *$users_{all}(v)$: all the transitive users of v*

For example, in Figure 1, $deps(a_2) = \{b_2, c_2\}$, $deps_{tree}(a_2) = \{b_2, c_2, d_1\}$, $users(d_1) = \{c_2, c_3\}$ and $users_{all}(d_1) = \{c_2, c_3, a_2\}$.

B. The Maven Library's Dependency Graph

In order to be able to reason about not only versions but also libraries, we elevate the abstraction of the MDG to the library level. Figure 2 shows the elevated graph corresponding to the dependency graph $\mathcal{G}$ in Figure 1. We construct a weighted graph, $\mathcal{G}_{\mathcal{L}}$, where nodes correspond to libraries (LIBS) in $\mathcal{G}$. We create an outgoing edge between two libraries l_1 and l_2 if there is at least a version of l_1 that uses a library version of l_2. We denote by $D(l)$ the set of direct library dependencies of a given library l. For example, $D(A) = \{B, C\}$. Finally, the weight of the outgoing edges from l_1 to l_2 corresponds to the number of versions of l_1 that use a version of l_2. We define the Maven Library's Dependency Graph (MDG_L) as follows:

Definition 3. ***Maven Library's Dependency Graph.*** *The MDG_L is a edge-weighted graph, where vertices represent Maven libraries, and edges' weight describes the number of dependency relationships between their versions. We define the MDG_L as $\mathcal{G}_{\mathcal{L}}$ = (LIBS, $\mathcal{E}$, $\mathcal{W}$), where,*

- *the set of vertices LIBS represents the libraries present in Maven Central*
- *the set of edges $\mathcal{E}$ represents the dependency relationships between libraries*
- *the weighing function $\mathcal{W}$ represents the weight of a given edge, defined as $\mathcal{W} : \mathcal{E} \rightarrow \mathbb{N}$*

For further references in the MDG_L, we introduce the following notations:

- *the set of direct library dependencies D of a given library, defined as D : LIBS $\rightarrow$ LIBS^n*
- *the weighing function $\overleftarrow{W}$ returns the sum of the weights of incoming edges, defined as $\overleftarrow{W}$: LIBS $\rightarrow \mathbb{N}$*
- *the weighing function $\overrightarrow{W}$ returns the sum of the weights of outgoing edges, defined as $\overrightarrow{W}$:LIBS $\rightarrow \mathbb{N}$*

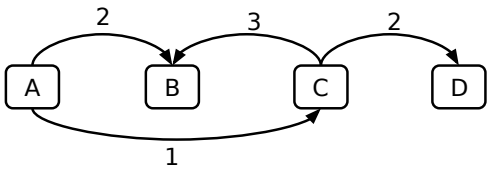

Fig. 2. The elevated Maven Library's Dependency Graph from Figure 1.

III. Study design

This work is articulated around five research questions. In this section, we introduce these questions as well as the metrics that we collect to answer them. We also describe the representative subset of artifacts that we study throughout the paper.

A. Research Questions

RQ1*: To what extent are the different library versions actively used?*

Because Maven artifacts are immutable, all the versions of a given library that have been released in Maven Central are always present in the repository. Meanwhile, previous studies have shown that users of a given version do not systematically update their dependency when a new version is released [5], [12], [13]. Consequently, we hypothesize that, at some point in time, multiple versions of a library are actively used. In this research question, we investigate how many versions are currently used, how many have been used but are not anymore and how many versions have never been used.

RQ2*: How are the actively used versions distributed along the history of a library?*

The full history of versions of a library released on Maven Central is always available. Consequently, users can decide to depend on any of the versions. In this research question, we analyze where, in the history of versions, are located the versions that are actively used.

RQ3*: Among the actively used versions of a library, is there one or several versions that are significantly more popular than the others?*

Library users are free to decide which version to depend on and for how much time. In the long term, these users' decisions determine what are the most popular libraries and versions in the entire software ecosystem [5], [14]. This research question investigates to what extent these decisions lead to the emergence of one or more versions that receive a greater number of usages compared to the other versions.

RQ4*: Does the number of actively used versions relate to the popularity of a library?*

We observe that for most libraries, more than one version is actively used at a given point in time. The library developers have no control over this since they cannot remove versions from Maven Central, nor force their users to update their dependencies. Meanwhile, it might be good for a library to maintain several versions that fit different usages. In this question, we investigate how the existence of multiple active versions relates to the overall popularity of a library.

RQ5*: How timely are the different library versions in Maven Central?*

With each new release, project maintainers make an effort to improve the quality of their libraries (e.g., by fixing bugs, adding new functionalities or increasing performance). These changes are expected to be directly reflected in the number of

users that update their dependencies to the new available release, and also in the number of new usages of the library [15]. This research question aims to get insights into how timely is the release of new versions. In particular, we investigate how much attraction gets a library version while it was the latest, compared to the older versions during the same period of time.

B. Metrics

To characterize the *activity status* of libraries and versions in terms of their usages by other latest library versions, we introduce the notions of *active*, *passive*, and *dormant* libraries and versions. Moreover, we introduce the *lifespan* of library versions to get insights on the duration of their activity period. These notions and measures are intended to answer RQ1 and RQ2.

Metric 1. ***Activity status.*** *A **passive library version** v is a version that has been used in the past, but is no longer used, even transitively, by any latest library version ($v \in$ LATESTS). Formally, this metric is described as a boolean function $isPsv : \mathcal{V} \rightarrow \{true, false\}$, where,*

$$isPsv(v) = \begin{cases} false & v \in \bigcup_{i \in LATESTS} \{deps_{tree}(i)\} \\ true, & otherwise \end{cases}$$

*An **active library version** v is a version where $isPsv(v) = false$. A **dormant library version** is an extreme case of a passive library version that occurs when the version has never been used by existing libraries (i.e., $users(v) = \emptyset$) in Maven Central.*

*At the library level, an **active library** is a library that has at least one active version, whereas a **passive library** is a library that has all its versions passive. A **dormant library** is an extreme case of passive library that occurs when all its versions are dormant.*

Metric 2. ***Lifespan.*** *The lifespan of a library version v is the time range during which it was/is being used. We define this period as the time range between the release date of v and the timestamp at which it becomes passive. In case v is active, this period starts at the release date of the artifact until the day the SNAPSHOT was captured. Dormant library versions do not have a lifespan at all. We denote this metric by $ls(v) = [startLs_v, endLs_v]$. Then, the interval's upper bound can be formally described as follows:*

$$endAct_v = \begin{cases} SNAPSHOT, & \neg isPsv(v) \\ last, & isPsv(v) \end{cases}$$

where, $last = \max \bigcup_{i \in users_{all}(v)} \{\mathcal{R}(next(i))\}$.

To study the *popularity* of library versions in Maven Central, and hence answer RQ3 and RQ4, we introduce a metric of popularity which measures the transitive influence and connectivity of a library version in the MDG. We rely on the standard PageRank algorithm [16], which accounts for the number of transitive usages. Intuitively, library versions with a higher PageRank are more likely to have a larger number of transitive usages. On the other hand, to measure the popularity of libraries, we use the Weighted PageRank algorithm [17] on the MDG_L.

Metric 3. ***Popularity.*** *The **popularity of a library version** $v \in \mathcal{V}$ is as follows:*

$$pop_{\mathcal{V}}(v) = (1-d) + d \sum_{i \in users(v)} pop_{\mathcal{V}}(i),$$

where d is a damping factor to reflect user behavior, which is usually set to 0.85 [18].

*The **popularity of a library** $l \in LIBS$ is as follows:*

$$pop_{\mathcal{L}}(l) = (1-d) + d \sum_{i \in U(l)} pop_v(i) \overleftarrow{c}_{(l,i)} \overrightarrow{c}_{(l,i)},$$

where $\overleftarrow{c}$ and $\overrightarrow{c}$ are respectively:

$$\overleftarrow{c}_{(l,i)} = \frac{\overleftarrow{W}(i)}{\sum_{p \in D(l)} \overleftarrow{W}(p)}, \quad \overrightarrow{c}_{(l,i)} = \frac{\overrightarrow{W}(i)}{\sum_{p \in D(l)} \overrightarrow{W}(p)}.$$

Finally, to answer RQ5, we introduce the notion of *timeliness* of library versions. This metric looks at the number of usages of every single version when it was latest and assesses if it was successful in attracting more users compared to its older versions. To this end, we compare the usages of a given version v during its lifespan to the usages that the whole library has received during the period when v was latest. We call this period the *timeliness period*.

Metric 4. ***Timeliness.*** *The **timeliness period**, $tp(v)$, of a library version v, is the time range between the release date of v and the most recently released version of its library ordered by $<_t$, which is not necessarily $next(v)$. We denote this version as mr:*

$tp(v) = [\mathcal{R}(v), \mathcal{R}(mr)]$,
where, $mr = \min_{i \in next_{all}(v)} \{\mathcal{R}(i) | \mathcal{R}(i) >_t \mathcal{R}(v)\}$.

*The **timeliness of a library version** v is a function, $tim(v) : \mathcal{V} \rightarrow \mathbb{Q}^+$, where,*

$$tim(v) = \frac{|users(v)|}{|\bigcup_{i \in \mathcal{V}} \{i | \mathcal{R}(i) \in tp(v) \wedge \mathcal{L}(v) \in \bigcup_{j \in deps(i)} \{\mathcal{L}(j)\}\}|}.$$

In case the library corresponding to v was not used during the timeliness period of v (the denominator is 0), then we consider $tim(v) = 0$. This also applies when v is dormant. All first releases of libraries have $tim(v) = 1$ since they have no earlier releases.

Based on the timeliness metric, three situations can occur:

- *v is **timely** if $tim(v) = 1$: v was a success during its timeliness period and users relied on it*
- *v is **over-timely** if $tim(v) > 1$: v has attracted users beyond its timeliness period*
- *v is **under-timely** if $tim(v) < 1$: users relied on older versions during its timeliness period*

TABLE I
CATEGORIES OF LIBRARIES IN MAVEN CENTRAL ACCORDING TO THEIR RELEASING PROFILES

Category	Criteria	#Libraries (%)	#Versions (%)
(i)	#versions = 1	65,557 (29.33%)	65,557 (2.72%)
(ii)	One shot*	32,825 (14.69%)	459,445 (19.08%)
(iii)	#versions > 1	125,096 (55.98%)	1,882,333 (78.2%)

(*) Libraries with more than one version and that have been released in the same day.

C. Study Subjects

During our initial exploration of the MDG, we distinguished three different categories of libraries in Maven Central: (i) libraries that have only one version ($\sim$30%), (ii) libraries with multiple versions all released on the same day ($\sim$15%), and (iii) libraries with multiple versions released within different time intervals ($\sim$55%). Table I gives detailed numbers about these categories. In particular, after manual inspection we notice that a large number of libraries belonging to categories (i) and (ii) are shipped with their classpath. We suspect these projects to be using Maven only for deploying and storing their libraries in Maven Central, but not for dependency management or further maintenance tasks.

In this work, we are interested in studying libraries that have multiple versions and utilize Maven regularly to manage and update their dependencies, i.e., libraries belonging to category (iii) in Table I. Figure 3 shows the distribution of the number of versions for the libraries in this category. The minimum and maximum number of versions are respectively 2 and more than 2,000, precisely, 2,122. Meanwhile, the 1st-Q and 3rd-Q are around 5 and 200 versions respectively.

In order to conduct our empirical study on a representative dataset, we choose [1st-Q, 3rd-Q] as a range of number of versions. Therefore, this study focuses on all the libraries with between 5 and 200 versions. This accounts for 73,653 libraries and 1,487,956 versions, representing 32.96% and 61.81% of the total number of libraries and version in Maven Central at the `SNAPSHOT` time.

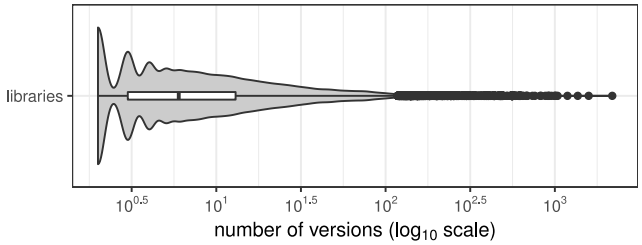

Fig. 3. Distribution of the number of library versions in Maven Central.

IV. RESULTS

In this section, we address our research questions and present the results obtained.

A. *RQ1: To what extent are the different library versions actively used?*

To answer RQ1, we study the activity status of libraries and versions in Maven Central. Table II shows the numbers and percentages of active, passive and dormant libraries and versions. We observe a low percentage of active versions (14.73%), whereas there is a predominant number of passive ones (85.27%), of which more than a half are dormant (45.16%). On the other hand, we can notice that the majority of libraries are active (95.49%), i.e., have at least one of its versions active. Meanwhile, passive libraries represent nearly 5% of the total number of libraries, of which ($\sim$4%) are dormant.

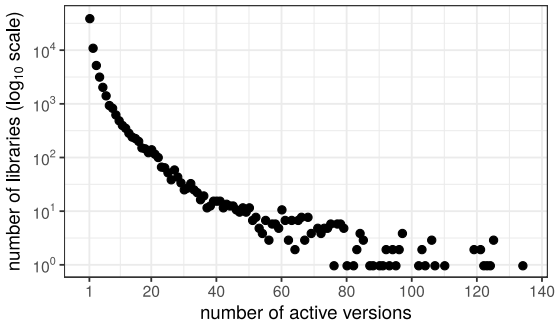

Fig. 4. Distribution of the number of active versions across active libraries

TABLE II
ACTIVITY STATUS OF LIBRARIES AND VERSIONS IN THE STUDY SUBJECTS

Status	#Versions (%)	#Libraries (%)
Active	219,184 (14.73%)	70,337 (95.49%)
Passive non-dormant	596,776 (40.11%)	387 (0.53%)
Dormant	671,996 (45.16%)	2,929 (3.98%)
Total	1,487,956 (100%)	73,653 (100%)

We are intrigued by the 2,929 dormant libraries. The median number of versions in this family of libraries is 9 with a maximum of 150 versions. We noticed that most of them are in-house utility libraries, intended for custom logging or testing, e.g., '*com.twitter:util-benchmark_2.11.0*'. Other libraries are archetypes[5], e.g., '*io.fabric8.archetypes:karaf-cxf-rest-archetype*'. These libraries are not intended to be used in production. Their custom nature makes them used rather internally, or by the library maintainers themselves.

In Table II, we also observe that a low proportion of versions are active 219,184 (14.73%), yet they are distributed across a very high number of libraries, 70,337 (95.49%), making these libraries active. Figure 4 summarizes the distribution of active versions in active libraries. We observe that more than a half of active libraries, 40,233 in total, have only one active version. The remainder, 30,104 libraries, have more than one active version. For some libraries, such as '*org.hibernate:hibernate-core*', more than 100 versions are currently active. However,

[5]https://maven.apache.org/guides/introduction/introduction-to-archetypes

the number of libraries with more than 100 active versions represents less than 2% of the total. More interestingly, we notice that 17% of the libraries have active versions belonging to more than one different major releases (e.g., 2.X.X). For instance, the library '*activemq:activemq*' has two active major versions: 3.X.X and 4.X.X, whereas '*com.spotify:docker-client*' has seven active versions: from 2.X.X up to 8.X.X.

Figure 5 shows the lifespan distribution of active and passive versions. To avoid the bias introduced by the SNAPSHOT time constraint, we consider only non-latest active versions of libraries ($v \notin$ LATESTS). As we can see from the figure, the lifespan of passive versions is approximately distributed between 8 and 80 days (1st-Q and 3rd-Q), whereas, this range is larger for active versions: between 351 and 1,626 days. This conveys that versions that are active for more than 80 days are likely to remain active for a longer period. Subsequently, these libraries are likely to be popular and widely used. Finally, we notice that the median number of days a version spends after its creation before being used for the first time is 14, with a mean of 57.61. This suggests that versions that have been dormant for less than 57 days are likely to become active; beyond this time period, they are likely to remain dormant.

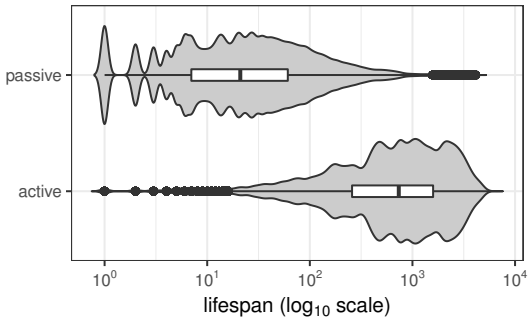

Fig. 5. Distributions of the lifespan (in days) of passive and active versions

Findings from RQ1: More than 40% of libraries in Maven Central have strictly more than one active version, while almost 4% of the libraries have never been used. This hints on an inclusive, immutable repository that can support the emergence of a diversity of library usages.

B. *RQ2: How are the actively used versions distributed along the history of a library?*

According to Metric 1, active libraries have at least one active version. In this research question, we focus on understanding how these active versions are distributed across the different library releases.

Figure 6 shows the positional distribution of all the active versions in the libraries. Since libraries can have different number of versions, we use a normalized relative index lying between $[0, 1]$, where 0 and 1 represent the indexes of the first and last versions of the library, respectively. First of all, we observe that active versions are scattered across different positional indices. While 68.4% of active library versions are almost evenly distributed across the non-latest releases, a significant number of active versions, precisely 69,146 (31.6%), are latest versions. This result is inline with the current policies of dependency management systems, which recommend upgrading to latest dependencies.

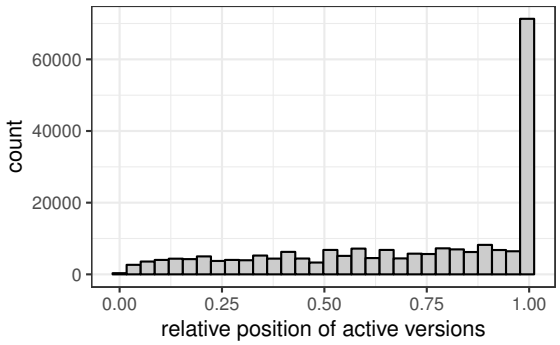

Fig. 6. Positional distribution of active versions (#bins = 30).

Digging further, we investigate the transitional distribution of active and passive versions. To do this, we transform each library $l \in$ LIBS into a vector, $\mathcal{S}_l$, capturing the passive/active status corresponding to all of its versions. Our objective is to analyze the occurrence of common transitional patterns between active and passive versions.

Let $\mathcal{S}_l$ be a vector representing the activity status of library versions ordered by $<_v$ (i.e. ordered by version number). The status corresponding to a version v is P if $isPSV(v)$ is $TRUE$ and A otherwise. For example, the library *com.google.guava:guava-jdk5* has a total of five versions, i.e., $\mathcal{S}_{guava-jdk5}$ = [A,A,P,P,A]. Considering that we are particularly interested in transitional patterns, the consecutive versions with the same status can be compressed to a single status, e.g., the previous example is represented as [A,P,A].

TABLE III
THE TOP-7 MOST COMMON TRANSITIONAL PATTERNS

Pattern	Frequency	Example
[P,A]	43,549	*commons-codec:commons-codec*
[P,A,P,A]	10,219	*org.apache.commons:commons-lang3*
[P,A,P,A,P,A]	3,478	*org.jboss.logging:jboss-logging*
[A,P, A]	2,761	*com.google.guava:guava-jdk5*
[P,A,P,A,P,A,P,A]	1,592	*org.joda:joda-convert*
[A,P,A,P,A]	1,343	*com.google.inject:guice*
[P,A,P]	613	*org.springframework:spring-webflow*

We obtained a total of 94 different transitional patterns. Table III shows the frequency of appearance of the seven most common of them. As expected, the 92% of the patterns are finishing by an A. The most frequent pattern is [P,A], i.e., old versions are passive and the latest ones are active. Yet, the remaining patterns represent more than 40% of the libraries. The rest of libraries follow a pattern where some old versions are also active. In extreme cases, the latest version of the library is passive (patterns finishing with a P). In such cases, we observe that most of their clients use an older version with the same major version number. We

speculate that this behavior is due to the clients' belief that the version they use is rather stable. Similar findings have been reported by Kula et al. [12]. We also observe that 5.5% of the libraries have their earliest version active. It is interesting to note that many of them are very popular libraries, e.g., '*org.hamcrest:hamcrest-core*' and '*org.apache.ant:ant*'.

Findings from RQ2: 31.6% of active versions are latest and the remaining 68.4% of active versions are evenly distributed across the libraries' history. When the clients do not use the latest version, they often depend on earlier versions belonging to the same major release of the library.

C. *RQ3: Among the actively used versions of a library, is there one or several versions that are significantly more popular than the others?*

In this research question, we investigate the diversity in the popularity of library versions. We assess the popularity of a library versions using Metric 3. In particular, we are interested in identifying significantly popular versions and analyzing the positional distribution of these versions. For this aim, we use the Tukey's outlier detection method [19] to identify versions with a popularity score that is far greater than the remaining versions of the library.

We distinguish between three different classes of libraries: (i) libraries that do not have a significantly most popular version (55,148), (ii) libraries with one significantly popular version (9,622), and (iii) libraries with more than one significantly popular version (8,883). The first class (i) represents libraries with versions that have a similar number of usages. The classes (ii) and (iii) represent libraries with one or more versions that have attracted more users compared to the rest of their versions. A large number of the users of significantly popular versions are different versions of the same library. These are library providers that may have remained loyal to one version despite the release of newer versions. To our surprise, almost all the significantly popular versions are active, only 86 out of 143,334 are passive. For instance, '*com.amazonaws:aws-java-sdk:1.11.409*' is significantly popular and passive.

Figure 7 shows illustrative examples, Apache IO, JUnit, and XML APIs, each one corresponding to one of these three classes. The horizontal dashed line in each frame represents the outlier's threshold of the library. All the versions that lie above this line are considered significantly popular. As shown in the figure, although the version 2.4 of Apache IO is quite old, it is still the most popular release of this library in Maven Central. In the case of JUnit, it has two significantly popular versions: 4.11 and 4.12. On the other hand, the library XML APIs does not have any significantly popular version (i.e., the popularity of its versions remains steady across time).

In order to measure the positional distribution of popular versions, we focus on libraries that have at least one significantly popular version. We determine the relative position of such versions with respect to the number of versions of the library. As for the positional distribution of active version, we also normalize the relative position between $[1, 0]$. The histogram in Figure 8 shows the distribution of the positions of the most popular versions across libraries. We observe that less than 10% of libraries have their latest version as the most popular. This is expected since the average lifespan of latest versions is lower than the average of non-latest versions. Interestingly, we found that the remaining highly popular versions are almost equally distributed, between 2% and 5%, in the remaining positions.

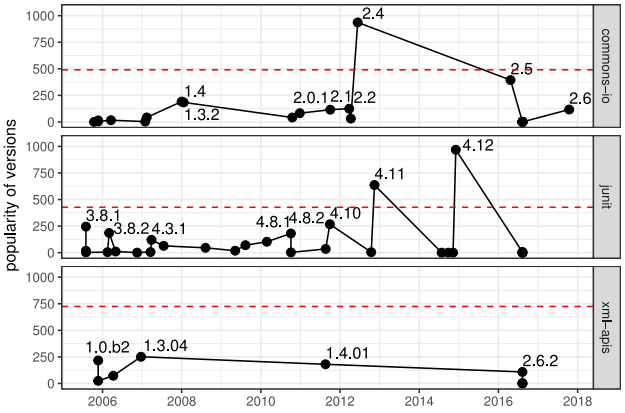

Fig. 7. Evolution of the popularity of versions ($pop_{\mathcal{V}}(v)$ metric) corresponding to the libraries Apache Commons IO, JUnit and XML APIs.

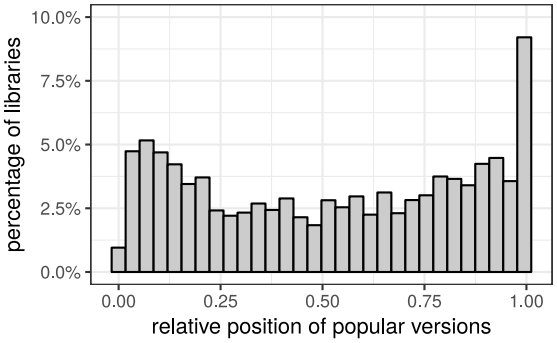

Fig. 8. Histogram of the positional distribution of significantly popular library versions (#bins = 30).

This result indicates that the most popular libraries in Maven Central are distributed across all the different library releases. It is notable that for almost 85% of libraries the most used version is not the latest. Thus, older versions are still being heavily used by other libraries, with the exception of the first version which is rarely the most popular.

Findings from RQ3: 17% of the libraries have more than one significantly popular version distributed across different releases, each of which creates a niche fitting a group of users. This indicates that library developers successfully address the needs of diverse populations of users.

D. *RQ4: Does the number of actively used versions relate to the popularity of a library?*

We have seen so far that many libraries in Maven Central have multiple active versions, of which more than one can be significantly more popular than the others. Now, we investigate whether the activity status of versions has a direct effect on the popularity of their corresponding library. For this, we calculate the percentages of active and passive versions of each library and compare them with respect to the overall popularity of the library.

Figure 9 shows the smoothing function corresponding to the relation between the popularity of libraries and their percentages of active versions. There is a significant positive correlation between both variables (Spearman's rank correlation test: $\rho = 0.87$, p-value < 0.01). In particular, we observe that libraries that have more than 50% of active versions are more likely to be very popular, as popular libraries with many versions attract more clients for their versions.

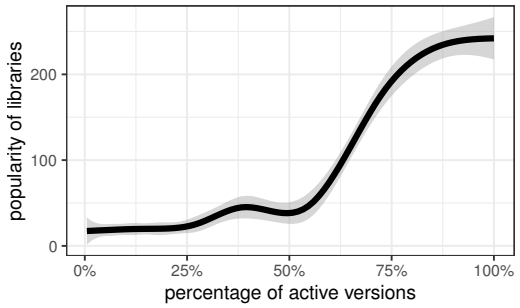

Fig. 9. Fitting curve (GAM model) of the percentage of active versions w.r.t. the popularity of libraries ($pop_{\mathcal{L}}(l)$ metric). The shaded area around the fitting curve represents the 95% confidence interval.

Table IV shows the seven most popular libraries ranked in decreasing order of popularity, as well as their percentage of active and significantly popular versions. As we can see, in all the cases a significant proportion of their versions are active. This indicates that many versions of these libraries continue being actively used, contributing to the popularity of the library and adding dependency diversity among all the clients. In three cases out of seven, there are more than two versions that are significantly more popular than the others. Finally, we also notice that these popular libraries serve general purposes, which allow them to fit well for various types of usages.

Findings from RQ4: Popular libraries in Maven Central have most of their versions active and serve general purposes. Moreover, the popularity of a library can be estimated by the number of its active versions. The more active versions a library has, the more likely it is to be popular, and vice-versa.

E. *RQ5: How timely are the different library versions in Maven Central?*

This research question focuses on the temporal dimension of the dataset. We analyze whether the diversity of popular and active versions that we observe today is a phenomenon that sustained in the past history of the libraries. We look at every single library version v separately and investigate whether, during the time period when v was the latest, it gained the expected attraction among its older peers. We compare the number of usages that a version v gets during its lifespan period against the number of usages that the whole library received during the timeliness period of v. For this comparison, we rely on the timeliness function described in Metric 4. This metric can be considered as an internal popularity metric that assesses the popularity of a version among its peers.

TABLE IV
THE TOP-7 MOST POPULAR LIBRARIES IN OUR STUDY SUBJECTS AND THEIR NUMBER OF ACTIVE AND SIGNIFICANTLY POPULAR VERSIONS

Library	Domain	#Active (%)	#Popular (%)
google.code.findbugs:jsr305	Utility	10 (90%)	1 (9.01%)
org.slf4j:slf4j-api	Logging	63 (86.3%)	3 (4.1%)
log4j:log4j	Logging	18 (94.7%)	1 (5.2%)
com.google.guava:guava	Utility	71 (79.7%)	1 (1.2%)
junit:junit	Testing	27 (96.5%)	2 (7.1%)
org.hamcrest:hamcrest-core	Testing	5 (100%)	1 (20%)
commons-logging:logging	Logging	15 (88.3%)	2 (11.8%)

Overall, for all our study subjects, 70.6% of library versions are under-timely (including dormant versions), while 19.8% are timely, and the remaining 9.6% are over-timely. Figure 10 shows the distribution of the three timeliness classes for active and passive versions. We observe that roughly 45% of passive library versions were under-timely. These are versions that did not attract users for their library throughout their timeliness period. Meanwhile, almost 55% are timely. These are library versions that were not only active at some point, but also widely used. This gives substantial evidence that the diversity that we observe today has existed in the past in Maven Central. On the other hand, we observe that 55.3% of active versions are under-timely. These are versions that are not widely popular among their peers, yet active. The average lifespan of these versions is $\sim$777 days, which suggests that although they are under-timely, they are likely to remain active for a long period of time; whereas, the remaining active versions are evenly distributed among timely and over-timely.

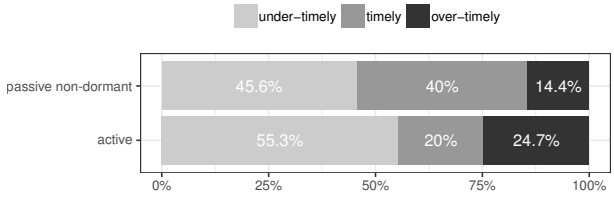

Fig. 10. Proportions of timeliness classes for passive and active versions.

In order to analyze the distribution of the timeliness classes at the library level, we calculate the proportions of under-timely, timely and over-timely versions in each library. Figure 11 shows a ternary diagram [20] representing the distribution of the three timeliness classes across the study subjects. In the figure, each point represents a library. In

general, we observe a high dispersion in the space of libraries, meaning that there are representative cases for almost all of the different proportions of classes. The paired correlation tests between the proportions of each of the classes and the popularity of their corresponding library reveal that none of the correlations are statistically significant (p-value > 0.05 according to the Spearman's test). Therefore, the proportions of the timeliness of the versions of a library are not directly related with the popularity of the library.

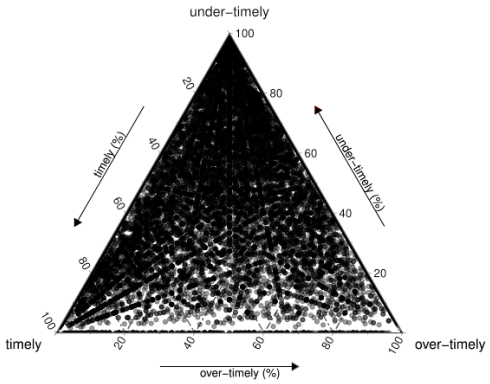

Fig. 11. Distribution of libraries w.r.t. their percentages of over-timely, timely and under-timely versions. The dispersion of points inside the triangle indicates that the proportions of classes are well distributed across the libraries.

Findings from RQ5: The diversity in the usage and popularity of versions has consistently sustained during the history of Maven Central. We observe that $\sim 10\%$ of all the library versions attract new users during their timeliness period and remain active even after the next version has been released. Meanwhile, there is no correlation between the popularity of a library and the timeliness of its versions.

V. Discussion

In this section, we discuss the implications of our findings about the emergence of software diversity in Maven Central, as well as some threats to the validity of our results.

A. Supporting the Emergence of Software Diversity

This study focuses on the diversity of usages of libraries and versions in Maven Central. We have observed empirically how the immutability of versions, which is a characteristic enforced by design in Maven Central, supports the natural emergence of software diversity [3]. This diversity takes multiple forms and has various effects:

- all active libraries have strictly more than one active version, and the 42.7% of them have more than two active versions;
- 17% of the libraries have two or more versions that are significantly more popular than the others, which indicates a very rich diversity in usages of the latest library releases and may imply that the latest library versions deployed on Maven Central use different versions of a similar library;
- the most popular libraries are also the ones that have the largest proportion of active versions;
- the existence of multiple used versions that overlap in time is a common phenomenon in the history of all libraries.

We interpret these multiple forms of diversity in usage and popularity of libraries as follows: a repository that offers the opportunity for users to choose their dependencies, naturally supports the emergence of diversity among these dependencies. In other words, this massive emergent diversity is not only due to users who forget to update their dependencies. Many users decide very explicitly to depend on one or the other version of a library because it perfectly fits their needs. Consequently, this kind of diversity emerges in a fully decentralized and unsupervised manner.

Our study also highlights some important challenges for a repository that supports diversification. First, there is a cost for the maintainers of Maven Central. We have observed that, although most libraries are actively used (95.49%), only 14.73% of the Maven artifacts are used. We have also noticed that some companies use Maven Central to store artifacts that nobody else uses (45.1% of versions are dormant). Consequently, keeping all versions induces an overhead in hardware and software resources. Second, there is a cost for the developers of popular libraries who need to maintain several versions of their library to serve different clients. Third, there is a risk that users decide to keep a dependency towards a vulnerable or flawed version.

The trade-off between healthy levels of diversity in a system (here, the Maven Central ecosystem) and the challenges of redundancy and noise is necessary and very natural. Biological studies insist on the importance of keeping less fit or even unexpressed genes as genetic material that is necessary in order to adapt to unpredictable environmental changes [21], [22]. Our study reveals that the immutability of Maven artifacts provides the material for libraries to eventually fit the needs of various users, which eventually results in the emergence of diverse popular and timely versions. In the same way that biological systems do, library maintainers can accommodate the overhead of manual updates and conflict management in order to contribute to the sustainability of the massively large pool of software diversity that exists in Maven Central.

B. Threats to Validity

We report about internal, external, and reliability threats to the validity of our study.

a) Internal validity: The internal threat relates to the metrics employed, especially those to compare the popularity of libraries and versions. In this work, we characterize popularity in terms of number of usages and quantify it based on well-known graph-based metrics [23]. Thus, we assume that a widely reused library is a popular one, and we consider only the relationships described in Maven Central, which do not take into account usages from private projects. The jOOQ library is one example among others. Because it is dual-license, many OSS libraries avoid to depend on it, but other

closed-source software are still using it and there is no way to quantify their number. However, as suggested in previous studies, software popularity can be measured in a variety of ways, depending on different factors such as social or technical aspects [24]. Another concern relates to the fact that conventions on semantic versioning are not really taken well into account by library maintainers [25]. Still, we believe that at the scale of the dataset employed in this study, our metrics are a fair approximation of the state of practice in Maven Central.

b) External validity: Our results might not generalize to other software repositories beyond the Maven Central ecosystem (e.g., npm, RubyGems or CRAN). It should also be noticed that Maven Central does not perform any real vetting of the people that deploy artifacts or on the quality of such artifacts. Thus, the integrity and origin of most of our study subjects therein is not known or verifiable. Moreover, this work takes into account version ordering as well as temporal ordering relationships, which we believe are sufficient to give a plausible representation of the way that libraries are updated as well as their evolution trends.

c) Reliability validity: Our results are reproducible, the dataset used in this study is publicly available online[6]. Moreover, we provide all necessary code[7] to replicate our analysis, including Cypher queries and R notebooks.

VI. Related work

This paper is related to a long line of previous works about mining software repositories and analysis of dependency management systems. In this section we discuss the related work along the following aspects.

a) Structure and updating behavior: Over the past years, several research papers have highlighted the benefits of leveraging graph-based representations and ecologycal principles to analyze the architecture of large-scale software systems [26]–[29]. Raemaekers et al. [30] investigated the adherence to semantic versioning principles in Maven Central as well as the update trends of popular libraries. They found that the presence of breaking changes has little influence on the actual delay between the availability of a library and the use of the newer version. Kula et al. [12] study the latency in trusting the latest release of a library and propose four types of dependency adoptions according to the dependency declaration time. De Castilho et al. [31] use the Maven Central repository for automatically selecting and acquiring tools and resources to build efficient NLP processing pipelines. Their analysis relied partially on Maven build files to collect library dependencies in industrial systems. However, as far as we know, none of the existing works have studied the repercussion of the artifacts' immutability at the scale of the entire Maven Central repository.

b) Analysis of evolution trends: The evolution of software repositories is a popular and widely-researched topic in the area of empirical software engineering. Recently, Decan et al. [32] perform a comparison of the similarities and differences between seven large dependency management systems based on the packages gathered and archived in the *libraries.io* dataset. They observe that dependency networks tend to grow over time and that a small number of libraries have a high impact on the transitive dependencies of the network. Kikas et al. [33] study the fragility of dependency networks of JavaScript, Ruby, and Rust and report on the overall evolutionary trends and differences of such ecosystems. Abdalkareem et al. [34] investigate about the reasons that motivate developers to use trivial packages on the npm ecosystem. Raemaekers et al. [35] construct a Maven dataset to track the changes on individual methods, classes, and packages of multiple library versions. Our work expands the existing knowledge in the area by showing how software repositories can contribute to prevent dependency monoculture by making available a more diverse set of library versions for software reuse.

c) Security and vulnerability risks: Researchers have investigated and compared dependency issues across many packaging ecosystems. Suwa et al. [11] investigate the occurrence of rollbacks during the update of libraries in Java projects. Their results confirm previous studies that show that library migrations have no clear patterns and in many cases, the latest available version of a library is not always the most used [36], [37]. Mitropoulos et al. [38] present a dataset composed of bugs reports for a total of $17,505$ Maven projects. They use FindBugs to detect numerous types of bugs and also to store specific metadata together with the FindBugs results. Zapata et al. [39] compare how library maintainers react to vulnerable dependencies based on whether or not they use the affected functionality in their client projects. Our work considers security and vulnerability risks in software repositories from a novel perspective, i.e., by taking into account the benefits and drawbacks that come with the emergence of software diversity.

VII. Conclusion

In this paper, we performed an empirical study on the diversity of libraries and versions in the Maven Central repository. We studied the activity, popularity and timeliness of $1,487,956$ artifacts that represent all the versions of $73,653$ libraries. We defined various graph-based metrics based on the dependencies among Maven artifacts that are captured in the Maven Dependency Graph [1]. We found that $\sim 40\%$ of libraries have two or more versions that are actively used, while almost 4% never had any user in Maven Central. We also found that more than 90% of the most popular versions are not the latest releases, and that both active and significantly popular versions are distributed across the history of library versions. In summary, we presented quantitative empirical evidence about how the immutability of artifacts in Maven Central supports the emergence of natural software diversity, which is fundamental to prevent dependency monoculture during software reuse. Our next step is to investigate how we can amplify this natural emergence of software diversity through dependency transformations at the source code level.

[6]https://doi.org/10.5281/zenodo.1489120

[7]https://github.com/castor-software/oss-graph-metrics/tree/master/maven-central-diversity

ACKNOWLEDGMENTS

This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529, by the Wallenberg Autonomous Systems and Software Program (WASP), by the TrustFull project financed by the Swedish Foundation for Strategic Research and by the OSS-Orange-Inria project.

REFERENCES

[1] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central," in *16th International Conference on Mining Software Repositories (MSR)*, (Montreal, Canada), IEEE/ACM, 2019.

[2] M. Stamp, "Risks of monoculture," *Commun. ACM*, vol. 47, pp. 120–, Mar. 2004.

[3] B. Baudry and M. Monperrus, "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond," *ACM Computing Survey*, vol. 48, no. 1, pp. 16:1–16:26, 2015.

[4] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the Dependency Conflicts in my Project Matter?," in *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 319–330, 2018.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Software Engineering*, vol. 23, pp. 384–417, Feb 2018.

[6] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable Open Source Dependencies: Counting Those That Matter," in *12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, (New York, NY, USA), pp. 42:1–42:10, ACM/IEEE, 2018.

[7] K. Jezek, J. Dietrich, and P. Brada, "How Java APIs Break – An Empirical Study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[8] J. H. Lala and F. B. Schneider, "It monoculture security risks and defenses," *IEEE Security & Privacy*, vol. 7, no. 1, pp. 12–13, 2009.

[9] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, pp. 280–294, Oct 2007.

[10] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds: Exploiting the intrinsic redundancy of web applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, pp. 16:1–16:42, May 2015.

[11] H. Suwa, A. Ihara, R. G. Kula, D. Fujibayashi, and K. Matsumoto, "An Analysis of Library Rollbacks: A Case Study of Java Libraries," in *24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pp. 63–70, Dec 2017.

[12] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, (Montreal, Canada), pp. 520–524, 2015.

[13] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, pp. 1275–1317, Oct 2015.

[14] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining Trends of Library Usage," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, (New York, NY, USA), pp. 57–62, ACM, 2009.

[15] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, "An Exploratory Study on Library Aging by Monitoring Client Usage in a Software Ecosystem," in *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 407–411, 2017.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, Stanford InfoLab, November 1999.

[17] W. Xing and A. Ghorbani, "Weighted PageRank Algorithm," in *Proceedings of the Second Annual Conference on Communication Networks and Services Research (CNSR)*, pp. 305–314, May 2004.

[18] P. Boldi, M. Santini, and S. Vigna, "PageRank As a Function of the Damping Factor," in *14th International Conference on World Wide Web (WWW)*, (New York, NY, USA), pp. 557–566, ACM, 2005.

[19] J. Tukey, "Exploratory data analysis," 1977.

[20] N. E. Hamilton and M. Ferry, "ggtern: Ternary diagrams using ggplot2," *Journal of Statistical Software*, vol. 87, no. CN 3, pp. 1–17, 2018.

[21] R. Frankham, "Genetics and extinction," *Biological Conservation*, vol. 126, no. 2, pp. 131–140, 2005.

[22] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4 + 1 popular java apis and the jdk," *Empirical Software Engineering*, vol. 23, pp. 2158–2197, Aug 2018.

[23] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components based on Use Relations," *IEEE Transactions on Software Engineering*, vol. 31, pp. 213–225, March 2005.

[24] A. Zerouali, T. Mens, G. Robles, and J. Gonzalez-Barahona, "On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019.

[25] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140 – 158, 2017.

[26] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, "Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases," in *""* (D. Winkler, S. Biffl, and J. Bergsmann, eds.), pp. 125–148, Springer, 2019.

[27] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions," in *21st International Conference on Automated Software Engineering (ASE)*, pp. 199–208, IEEE/ACM, Sep. 2006.

[28] T. Mens, M. Claes, and P. Grosjean, "ECOS: Ecological Studies of Open Source Software Ecosystems," in *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 403–406, Feb 2014.

[29] T. Mens and P. Grosjean, "The ecology of software ecosystems," *Computer*, vol. 48, pp. 85–87, Oct 2015.

[30] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in *14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 215–224, Sept 2014.

[31] R. E. de Castilho and I. Gurevych, "A broad-coverage collection of portable nlp components for building shareable analysis pipelines," in *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pp. 1–11, 2014.

[32] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, pp. 1–36, 2018.

[33] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," in *14th International Conference on Mining Software Repositories (MSR)*, (Buenos Aires, Argentina), pp. 102–112, IEEE/ACM, May 2017.

[34] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, (New York, NY, USA), pp. 385–395, ACM, 2017.

[35] S. Raemaekers, A. van Deursen, and J. Visser, "The Maven Repository Dataset of Metrics, Changes, and Dependencies," in *10th IEEE Working Conference on Mining Software Repositories (MSR)*, (San Francisco, CA, USA), pp. 221–224, IEEE, ACM, 2013.

[36] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A Study of Library Migrations in Java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

[37] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining Library Migration Graphs," in *19th Working Conference on Reverse Engineering (WCRE)*, (Kingston, Canada), pp. 289–298, October 2012.

[38] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The Bug Catalog of the Maven Ecosystem," in *11th Working Conference on Mining Software Repositories (MSR)*, (New York, NY, USA), pp. 372–375, ACM, 2014.

[39] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages," in *34th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 559–563, IEEE, 2018.